Substrate Code Review Feedback

Prepared By: BlockDeep Labs
Prepared For: Frequency Protocol

Team: Amplica Labs
Review Date: 12-08-2024
Report Date: 19-08-2024

Commit Hash: 9ef58185f2230aaf92dc166fb8eade56a78e4cf3
Review Scope Items:

- Passkey Pallet:
  - https://github.com/frequency-chain/frequency/tree/audit-2024/pallets/passkey
  - https://docs.google.com/document/d/1_HOKglyZeg0l_NPfFGHIexgFi86uRYEqp-4YwGYHbyE/edit#heading=h.fciz9jhoddkx
- Provider Boosting:
  - https://github.com/frequency-chain/frequency/pull/1694
  - https://docs.google.com/document/d/11iMT_47pY7fzYlM_HohqHgefjWBr_19avS2WmwTqwZA/edit?usp=sharing

Disclaimer:

This Code Review Report is provided by BlockDeep Labs without warranties of any kind, express or implied, as to the completeness, accuracy, reliability, or suitability of the reviewed code. The review is based on the information and code provided at the time of review and may not identify all potential vulnerabilities or issues.

BlockDeep Labs accepts no liability for any security breaches, losses, damages, or other consequences that may result from the deployment of the reviewed code on any network. The responsibility for ensuring the security and integrity of the code lies solely with the project team, who are advised to conduct further testing and audits as necessary.

Reviewer Feedback:

PassKey Pallet

1. **Benchmarks:**
   In a worst-case scenario, you might want to consider:

- **Large payloads:** The payload size could affect the time it takes to validate and dispatch it. You could create payloads of varying sizes and measure the time it takes to process them.

- **Invalid payloads**: You could measure how the system handles payloads that fail validation. This could include payloads with invalid signatures, nonces, or other parameters.

2. **Code Documentation:** While there is some documentation present, it could be more comprehensive. Each function should have a comment explaining what it does, its parameters, and its return value. For ex: At PasskeyNonceCheck struct definition there are few functions implemented that don't have any documentation.

3. **Use of Constants:** In benchmarking.rs, the `generate_payload` function uses hard-coded values for the account balance and the secret key. It would be better to define these as constants at the beginning of the module or in a separate constants file.

4. **Modularization:** The lib.rs file is quite long and contains many different components, including the pallet configuration, the pallet implementation, and several helper structs. It might be beneficial to break this file down into smaller, more manageable modules. For example, the `PasskeySignatureCheck`, `PasskeyNonceCheck`, and `ChargeTransactionPayment` structs could each be in their own file.

5. **Redundant Cloning:** There are several places in the code where values are cloned unnecessarily. For example, in the PasskeyNonceCheck and PasskeySignatureCheck structs, the validate and pre_dispatch methods clone the account_id (line 250) and passkey_call (line 276) structs. Consider passing these structs by reference to avoid unnecessary cloning.

Capacity Pallet

1. **Use of Constants:** In pallet lib.rs, the STAKED_PERCENTAGE_TO_BOOST constant is defined as 50. It would be better to define this as a configurable parameter in the Config trait and let runtime decide the value of this parameter.

2. **Code Duplication:** There seems to be some code duplication in the tests. For example, the setup code for creating a provider and staking tokens is repeated in several tests(for ex: unstake_tests.rs). This could be refactored into a helper function to reduce code duplication.

3. **Code Documentation:** While there is some good code documentation present, it could be more comprehensive. Each function should have a comment explaining what it does, its parameters, and its return value. For ex: do_withdraw_unstaked, reduce_capacity, start_new_reward_era_if_needed etc.

4. **Naming Conventions:** The naming convention could be improved. For example, in the `StakingDetails` struct, the variable `active` could be named more descriptively, similarly, the `new_active` variable in the `withdraw` function and `MaximumCapacity` in StakingType should follow a clearer naming convention.

5. **Error Handling:** The error handling in the code could be enhanced. For instance, there are several functions currently that return Option<()> or Result<(), ArithmeticError> (ex: do_retarget, increase_stake_and_issue_capacity). To provide more detailed feedback on errors, it would be beneficial to define a custom error type. This would allow for more informative error messages, particularly in scenarios where multiple operations occur consecutively. By identifying the exact stage, troubleshooting would become more straightforward.

Assessment:

Documentation and Codebase Quality:	**Feedback:**
a. Is there sufficient documentation for compiling and executing the code?	The documentation is sufficient.
b. Does the project documentation provide a conceptual understanding of the project?	While the project documentation is adequate, the detailed code documentation could benefit for each function to enhance understanding.
c. Are there written tests for the codebase, and do they run successfully, indicating completed code components?	There are many tests written with overall sufficient coverage, all of them were passed.

Code Review Comments:

(Subjective comments from engineers, similar to findings section in an audit report)

- Please ensure that the default is defined here.
- Also here.
- Maybe a more efficient implementation of this can be achieved with `try_mutate`

Grading:

Scope Item 1: Passkey Pallet

Topic	Grade	Feedback
Documentation	4	More Code documentation could be better.
Code Structure	4	Consider organizing different components into separate modules to improve structure and maintainability.
Code Performance	5	No issues found.
Tests	5	No issues found.

Scope Item 2: Capacity Pallet

Topic	Grade	Feedback
Documentation	4	Qualitative feedback TBC
Code Structure	5	Consider organizing different components into separate modules to improve structure and maintainability.
Code Performance	5	No issues found.
Tests	5	No issues found.

Supporting Reference for Assessment & Grading:

Criteria:

a) **Deliverable Functionality:** Assess whether the delivered code fulfills its intended purpose, a crucial aspect of evaluation.
b) **Documentation Quality:** Evaluate the comprehensiveness and clarity of documentation accompanying the delivery, aiming for accessibility to the broader community.
c) **Code Structure and Readability:** Examine the layout and clarity of the codebase to ensure it is understandable for community members who may review it.
d) **Performance Optimization:** Investigate the efficiency and performance of the chain, emphasizing avoidance of redundancies and straightforward implementation without the need for extensive benchmarking.
e) **Testing Coverage:** Emphasize the importance of thorough testing by evaluating the extent to which the project incorporates comprehensive testing protocols.

Grading: Scoring 1-5.

Documentation:

- **1) Inadequate:** Absence of documentation.
- **2) Deficient:** Documentation is present but lacks essential components.
- **3) Fair:** Existing documentation poses challenges in comprehension.
- **4) Adequate:** Documentation is comprehensive yet demands moderate effort to grasp.
- **5) Exemplary:** Documentation is impeccably structured, comprehensive, and easily comprehensible.

Code Structure:

- **1) Inadequate:** Code structure is convoluted and difficult to follow.
- **2) Deficient:** Some attempt at segmentation based on objectives, but lacks overall coherence.
- **3) Satisfactory:** Broad segmentation exists, however, with deficiencies in detailed structuring.
- **4) Proficient:** Segmentation of processes and tasks demonstrates good practice, albeit with minor shortcomings.
- **5) Exceptional:** Code division and structure are flawless, facilitating ease of comprehension and replication.

Code Performance:

- **1) Deficient:** Code performance is hindered by extensive repetitive patterns and lack of segmentation.
- **2) Subpar:** Notable redundancy exists, particularly within complex sections of the codebase.
- **3) Acceptable:** Segmentation is apparent, with discernible efforts to minimize redundancy, albeit not fully achieved.

- **4) Competent:** Overall execution is commendable, with minor areas identified for optimization.
- **5) Outstanding:** Code execution demonstrates exceptional performance, characterized by minimal redundancy and optimal efficiency.

Tests:

- **1) Deficient:** Testing procedures are entirely absent.
- **2) Limited:** Minimal testing efforts are evident, often simplistic in nature.
- **3) Adequate:** Testing coverage is satisfactory, though certain scenarios remain untested.
- **4) Proficient:** Testing protocols cover a majority of components adequately, with some areas identified for enhancement.
- **5) Exemplary:** Comprehensive testing which encompasses all conceivable scenarios, reflecting exemplary adherence to testing standards.